Kapitola: Ako si spraviť vlastný fixed-point combinator

1.1 Technické detaily o Pythone

Konštrukciu budeme robiť v Pythone, lebo je na naše účely dostatočne pohodlný. Malo by ale byť zjavné, ako podobné konštrukcie spraviť v iných jazykoch.

1.1.1 None

Konštanta `None` je špeciálna konštanta predstavujúca nič. Budeme ju používať v situáciách, kedy sme vo formalizme používali $\perp$, teda vtedy, keď mal program bežať do nekonečna. (V našich konkrétnych prípadoch si to môžeme dovoliť a `None` sa lepšie vypisuje.)

Pri niektorých operáciách budeme chcieť násobiť hodnoty, dorobíme si preto pomocnú funkciu, ktorá násobí, pričom ak je niektorý vstup `None`, tak vráti `None`:

```
def safe_mul(x,y):
    if x is None or y is None: return None
    return x*y
```

1.1.2 Funkcie s neznámym počtom argumentov

V Pythone vieme deklarovať funkciu, ktorá bude použiteľná s rôznymi počtami argumentov. Tu je základná syntax:

```
def f(x, *args):
    print( 'x = {}'.format(x) )
    print( 'dalsie parametre:', end='' )
    for y in args: print ( ' {},'.format(y) , end='' )
    print()

f(4)
f(4,7)
f(4,1,2,'zajac',print)
```

Toto bude pohodlné pri záverečnej konštrukcii, keď zostrojíme jediný funkcionál fungujúci naraz pre funkcie všetkých arít.

1.1.3 Výroba nových funkcií za behu

V Pythone vieme priamo manipulovať s funkciami a priraďovať si ich do premenných, napr. takto:

```
def add(x,y): return x+y

scitaj = add
vypis = print
vypis( scitaj(4,7) )
```

Toto sa nám zíde na stručný zápis niektorých konštrukcií.

1.1.4 Výroba nových funkcií pomocou reťazcov

Všetky konštrukcie, ktoré budeme robiť, by však išli robiť aj v ľubovoľnom inom jazyku, v ktorom vieme napísať jeho vlastný interpreter. Napr. v Pythone priamo existuje funkcia **exec**, ktorej dáme string obsahujúci program a ona ho vykoná. Takéto stringy môžeme samozrejme za behu meniť a vyrábať si tak nové programy.

```
# do premennej "program" si ulozime program funkcie faktorial
program = """
def f(n):
    if n==0: return 1
    return n * f(n-1)
"""
```

```
# pomocou volania exec vieme tento program spustit a vypisat si hodnotu 10!
exec( program + 'print(␣f(10)␣)' )

# zo stareho programu vieme lahko vyrobit novy, pocitajuci fciu g(n)=f(n)+7
novy_program = """
def g(n):
""" + program.replace('\n','\n␣␣␣␣') + """
    return f(n)+7
"""

# pozrieme si, co sme vlastne vyrobili
print(novy_program)

# pomocou volania exec vieme aj tento novy program spustit a vypisat si hodnotu 10!
exec(novy_program + 'print(␣g(10)␣)' )
```

1.2 Rekurzívna funkcia ako pevný bod funkcionálu

Všetci už isto vieme používať rekurziu a pomocou nej napríklad definovať faktoriál:

```
def faktorial(n):
    if n==0: return 1
    return n*faktorial(n-1)

print( [ faktorial(x) for x in range(10) ] )
```

Samotný faktoriál však vieme definovať aj ináč. Podľa rekurzívnej definície faktoriálu môžeme napísať nasledovný funkcionál `lepsi_faktorial`:

```
def lepsi_faktorial(nejaka_fcia):
    def nova_fcia(n):
        if n==0: return 1
        return safe_mul( n, nejaka_fcia(n-1) )
    return nova_fcia
```

Keď do funkcionálu `lepsi_faktorial` vopcháme nejakú funkciu, na výstupe nám vypadne nová funkcia (s o niečo dlhším programom), ktorá faktoriál ráta o chlp lepšie.

Môžeme si napríklad zobrať funkciu, ktorá nevie nič:

```
def nic_neviem(n): return None
print( [ nic_neviem(x) for x in range(10) ] )
```

Keď na tú použijeme náš funkcionál, dostaneme novú funkciu, ktorá už správne vypočíta 0!.

```
skoro_nic_neviem = lepsi_faktorial( nic_neviem )
print( [ skoro_nic_neviem(x) for x in range(10) ] )
```

Prípadne môžeme použiť náš funkcionál ešte $5\times$ a nová funkcia už bude vedieť rátať všetko po 5!.

```
for i in range(5): skoro_nic_neviem = lepsi_faktorial(skoro_nic_neviem)
print( [ skoro_nic_neviem(x) for x in range(10) ] )
```

No a pointa celého cirkusu: na funkciu faktoriál sa môžeme dívať ako na *pevný bod* nášho funkcionálu `lepsi_faktorial`. Faktoriál je teda taká funkcia, ktorej výstupy sa nezmenia, keď na ňu použijeme náš funkcionál.

Formálne, faktoriál je taká funkcia `f`, že na každom možnom vstupe `n` platí `f(n) == lepsi_faktorial(f)(n)`.

1.3 Samotná konštrukcia

Cieľom našej konštrukcie bude nájsť postup, ako program konkrétneho funkcionálu (napr. nášho `lepsi_faktorial`) **bez použitia rekurzie** vyrobiť program jeho pevného bodu (napr. faktoriálu). Skôr, než začneme, všimnite si, že náš funkcionál žiadnu rekurziu neobsahuje.

Dôsledkom konštrukcie bude nasledovné pozorovanie: V každom (ľubovoľne silnom) programovacom jazyku, v ktorom vieme interpretovať jeho vlastné programy, platí, že možnosť používať rekurziu nepridáva žiadnu novú výpočtovú silu. Všetko, čo vieme naprogramovať pomocou rekurzie, ide spraviť aj bez nej.

1.3.1 Faktoriál bez použitia rekurzie

Samozrejme, naprogramovať faktoriál bez použitia rekurzie je ľahké, stačí na to obyčajný cyklus. My si ale teraz ukážeme zvláštnejší spôsob, ktorý nebude špecifický pre faktoriál – pôjde teda zovšeobecniť.

Začneme nasledovnou zvláštnou funkciou:

```
def zvlastna_funkcia(funkcia,n):
    if n==0: return 1
    return n * funkcia(funkcia,n-1)
```

Čože je toto za zvieratko? Dostane dva vstupy, na prvý sa díva ako na funkciu, na druhý ako na číslo. Ak je to číslo 0, vráti na výstupe 1, inak zavolá tú funkciu, ktorú dostal na vstupe a dá jej dva parametre: ju samú a $n-1$. Jej výstup potom prenásobí číslom n a je hotovo.

Treba si všimnúť, že nikde tu nie je samotná rekurzia: definujeme jedinú funkciu `zvlastna_funkcia` a tá sama seba nikde nevolá.

Ak sa vám zdá podozrivé to volanie funkcie, ktorú sme dostali na vstupe, predstavte si namiesto neho príslušné volanie **`exec`**. Tam by to celé vyzeralo takto:

```
def zvlastna_funkcia_znovu(retazec_s_programom,n):
    if n==0: return 1
    nazov_funkcie = precitaj( retazec_s_programom )
    exec( retazec_s_programom + '\n' + 'vystup = ' + nazov_funkcie + '(' + retazec_s_programom + ',' + str(n) + ')' )
    return n * vystup
```

Zoberieme teda `retazec_s_programom`, ktorý sme dostali, ten prečítame a zistíme, aký je `nazov_funkcie`, ktorú tento program počíta. Následne zostrojíme nový reťazec a na ten zavoláme už existujúcu funkciu **`exec`**. My teda nikde žiadnu rekurziu nepoužívame.

No a ako zostrojiť faktoriál? Nič jednoduchšie. V tomto okamihu už existuje konkrétna funkcia `zvlastna_funkcia`, ktorú sme pred chvíľou naprogramovali. Môžeme jej teda na vstup ako prvý parameter dať ju samú.

```
print( [ zvlastna_funkcia(zvlastna_funkcia,x) for x in range(10) ] )
```

A prípadne pomocou nej definovať aj faktoriál ako samostatnú funkciu:

```
def zvlastny_faktorial(x): return zvlastna_funkcia(zvlastna_funkcia,x)
```

1.3.2 Faktoriál z jeho funkcionálu

Keď sa pozornejšie zahľadíme na funkciu `zvlastna_funkcia`, isto nám neunikne istá syntaktická podobnosť medzi ňou a našim starým známym: funkcionálom `lepsi_faktorial`. Mohlo by teda ísť vziať náš konkrétny funkcionál a čisto syntaktickými operáciami z neho vyrobiť zvláštnu funkciu podobnú tej z predchádzajúcej časti.

Toto celé teda chceme spraviť bez toho, aby sme sa museli zamýšľať nad tým, čo vlastne náš funkcionál robí. Len zoberieme jeho program aj s chlpami a vyrobíme z neho program príslušnej zvláštnej funkcie.

Ako bude tá vyzerať? Začneme od funkcie, ktorá dostane dva parametre – funkciu f a číslo n – a jediné, čo spraví, je, že zavolá f na vstupe (f, n). Keby sme takejto funkcii dali na vstup ako prvý parameter ju samú (teda jej program), veľa by toho nespočítala: pre ľubovoľné n by sa výpočet do nekonečna cyklil.

Teraz však príde k slovu náš funkcionál. Namiesto toho, aby sme zavolali tú funkciu f, ktorú sme dostali na vstupe, najskôr z nej naším funkcionálom vyrobíme novú funkciu, a až tú potom zavoláme samú na seba. Táto nová funkcia už nejaké vstupy vie vyriešiť sama od seba (toto vylepšenie pridal náš funkcionál) a pre tie ostatné spraví to isté ako tá stará funkcia – teda znovu vylepší svoj program použitím funkcionálu, a novú funkciu zavolá na nej samej. A tak ďalej, až kým sa po niekoľkých kolách nedopracujeme ku funkcii natoľko vylepšenej, že už náš vstup n bude vedieť spracovať.

Samotná konštrukcia si vyžaduje jeden technický krok. Náš funkcionál vie totiž vylepšovať len funkcie, ktoré spracúvajú iba relevantné vstupy, v našom prípade teda len unárne funkcie. Z funkcie, ktorá ako prvý parameter očakáva program, by vyrobil guláš. Takže funkciu f, ktorú dostaneme na vstupe, najskôr upravíme na unárnu funkciu a až tú potom vylepšíme.

```
def druhy_zvlastny_faktorial(funkcia,n):
    def vylepsi(fun):
        def aplikovana_funkcia(x):
            return fun(fun,x)
        return lepsi_faktorial(aplikovana_funkcia)
    return vylepsi(funkcia)(n)
```

Toto je ono. Pozrime sa, ako bude prebiehať výpočet pre $n = 3$.

Začneme teda volaním `druhy_zvlastny_faktorial(druhy_zvlastny_faktorial,3)`.

Pri vykonávaní tohto volania najskôr zostrojíme funkciu `vylepsi( druhy_zvlastny_faktorial )`.

Ako vznikne táto? Vyrobíme unárnu funkciu `aplikovana_funkcia(x)`, ktorá zavolá našu pôvodnú funkciu `druhy_zvlastny_faktorial` na vstupoch `(druhy_zvlastny_faktorial,x)`. No a túto aplikovanú funkciu následne vylepšíme naším funkcionálom a vylepšenú funkciu vyhodnotíme pre $n = 3$.

Keďže $3 \neq 0$, výstupom bude hodnota `safe_mul( n, aplikovana_funkcia(n-1) )`, čo je to isté ako `n * druhy_zvlastny_faktorial(druhy_zvlastny_faktorial,2)`.

A ajhľa. Skutočná rekurzia nikde, v princípe len manipulujeme s reťazcami, a predsa sa nám podarilo dosiahnuť rovnaký efekt. Ďalší priebeh výpočtu už je zjavný.

1.3.3 Ľubovoľná funkcia z príslušného funkcionálu

To, čo sme práve predviedli s funkcionálom `lepsi_faktorial`, ide zjavne spraviť s ľubovoľným iným konkrétnym funkcionálom. A dokonca vieme pridať úroveň abstrakcie: spravíme si na to funkciu `ozvlastni` (ktorá tiež bude funkcionálom).

Táto funkcia `ozvlastni` bude fungovať nasledovne: na vstupe do nej dáme funkcionál a ona nám z neho vyrobí a na výstupe vráti funkciu analogickú funkcii `druhy_zvlastny_faktorial`.

```
def ozvlastni(funkcional):
    def zvlastna_funkcia(funkcia,*args):
        def vylepsi(fun):
            def aplikovana_funkcia(*brgs):
                return fun(fun,*brgs)
            return funkcional(aplikovana_funkcia)
        return vylepsi(funkcia)(*args)
    return zvlastna_funkcia
```

No a teraz môžeme zobrať ľubovoľný funkcionál a použitím `ozvlastni` z neho vyrobiť zvláštnu nerekurzívnu funkciu, ktorá počíta to isté, ako jeho pevný bod:

```
treti_zvlastny_faktorial = ozvlastni( lepsi_faktorial )
print( [ treti_zvlastny_faktorial(treti_zvlastny_faktorial,x) for x in range(10) ] )
```

Aby sme explicitne zostrojili pevný bod daného funkcionálu, stačí výstup funkcie `ozvlastni` zabaliť do funkcie, ktorá ho spustí samého na sebe:

```
def y_kombinator(funkcional):
    zvlastna_fcia = ozvlastni(funkcional)
    def spravna_fcia(*args):
        return zvlastna_fcia(zvlastna_fcia,*args)
    return spravna_fcia
```

A tým sme dosiahli náš „svätý grál“: máme funkcionál `y_kombinator`, do ktorého vložíme program funkcionálu a on nám vyrobí program jeho pevného bodu. To všetko bez potreby rekurzie.

```
nerekurzivny_faktorial = y_kombinator( lepsi_faktorial )
print( [ nerekurzivny_faktorial(x) for x in range(10) ] )
```

1.3.4 Ďalšie príklady

Celé to samozrejme funguje aj pre rekurziu robiacu viac ako jedno rekurzívne volanie:

```
def lepsi_fibonacci(nejaka_fcia):
    def nova_fcia(x):
        if x<2: return x
        return nejaka_fcia(x-1) + nejaka_fcia(x-2)
    return nova_fcia

fibonacci = y_kombinator( lepsi_fibonacci )
print( [ fibonacci(x) for x in range(10) ] )
```

A takisto pre rekurzívne funkcie s viac ako jedným parametrom:

```
def lepsi_add(nejaka_fcia):
    def nova_fcia(x,y):
        if x==0: return y
        return 1+nejaka_fcia(x-1,y)
    return nova_fcia

add = y_kombinator( lepsi_add )
print( [ [ add(x,y) for y in range(5) ] for x in range(5) ] )
```

Aj ak sa rekurzia deje naraz na viac parametroch:

```
def lepsi_binomial(nejaka_fcia):
    def nova_fcia(n,k):
        if k<0 or k>n: return 0
        if k==0 or k==n: return 1
        return nejaka_fcia(n-1,k-1) + nejaka_fcia(n-1,k)
    return nova_fcia

binomial = y_kombinator( lepsi_binomial )
print( [ [ binomial(n,k) for k in range(n+1) ] for n in range(10) ] )
```

Vôbec netreba, aby parameter rekurzie klesal, stačí, keď pre naše vstupy výpočet časom skončí:

```
def lepsia_haluz(nejaka_fcia):
    def nova_fcia(n):
        if (n & (n-1)) == 0: return 0   # n==0 alebo n je mocninou 2
        return 1+nejaka_fcia(n+1)       # zavolame sa na *vacsiu* hodnotu
    return nova_fcia

haluz = y_kombinator( lepsia_haluz )
print( [ haluz(n) for n in range(20) ] )
```

1.3.5 Elegantný trik na záver

Rekurzívne funkcie sa zle debugujú. Ak by sme chceli, aby funkcia zapisovala svoje spustenie do logu, nie je zrovna najlepší nápad pridať ten zápis do nej. Napríklad nasledovná funkcia by nám asi príliš radosť nerobila:

```
def faktorial(n):
    print('spustil sa faktorial')
    if n==0: return 1
    return n*faktorial(n-1)
```

Totiž ak zavoláme `faktorial(40)`, dostaneme v logu krásnych 41 záznamov o spustení tejto funkcie, zatiaľ čo nám by bohate stačil ten jeden prvý. Na to by bolo treba spraviť wrapper:

```
def faktorial(n):
    if n==0: return 1
    return n*faktorial(n-1)

def logujuci_faktorial(n):
    print('spustil sa faktorial')
    return faktorial(n)
```

a následne samozrejme všade v kóde treba volať tento wrapper namiesto pôvodnej funkcie.

To zato keď si faktoriál vyrobíme y-kombinátorom z jeho funkcionálu, nie je nič ľahšie ako pri výrobe pridať takéto logovanie.

```
def logujuci_y_kombinator(funkcional):
    zvlastna_fcia = ozvlastni(funkcional)
    def spravna_fcia(*args):
        print('pouziva sa pevny bod pre', funkcional.__name__)
        return zvlastna_fcia(zvlastna_fcia,*args)
    return spravna_fcia
```

Stačí si faktoriál vyrobiť týmto novým fixed-point kombinátorom a logovanie je na svete. A keď sa pridá tento výpis do pôvodného y-kombinátora, vieme naraz zapnúť/vypnúť logovanie pre všetky naše rekurzívne funkcie.

Ten istý efekt ide samozrejme v rôznych situáciách porovnateľne pohodlne dosiahnuť aj ináč, ale nový uhol pohľadu nikdy nie je na škodu.

Alebo naopak. Máte knižnicu a v nej rekurzívnu funkciu. Chceli by ste logovať úplne všetky volania – ale to nejde, ak knižnicu nemôžete meniť. Tu vám už wrapper nepomôže. Ak by ale v knižnici bol namiesto rekurzívnej funkcie príslušný funkcionál, nič vám nebráni vyrobiť si z neho dotyčnú funkciu ktorá naozaj zaloguje každé svoje volanie.

Ďalší benefit: z funkcionálu v prípade potreby vieme ľahko vyrobiť aj memoizovanú verziu dotyčnej rekurzívnej funkcie.